Lambda expressions in Fortran

Arjen Markus
arjen.markus@deltares.nl

May 30, 2019

1 Introduction

Many modern programming languages have a feature called *lambda expressions* that allows the programmer to pass simple expressions as *arguments* to some subprogram instead of having to write a subprogram that contains the expression. Here is an example in Java:[1]

```
printPersons(
    roster,
    (Person p) -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);
```

The idea is that the expression `(Person p) -> ...` is more or less inserted into the function `printPersons` where it is used to select only those persons from the list contained in the object `roster` that satisfy the criteria. Other uses could be to define some arithmetic function on the fly, as it were, instead of a full-fledged Java class and method.

The question discussed in this short note is: can we do the same in Fortran or would we need some new features and accompanying syntax (like the `->` operator in the Java example)? The quick answer is: we can get quite close with what is already available in Fortran – even if we restrict ourselves to the Fortran 90 standard. The only drawback from that latter restriction is that certain syntactic details are unavailable.

A simple application of the `lambda_expressions` module looks like this:

```
program print_table
    use lambda_expressions

    type(lambda_integer)           :: x
    type(lambda_expression)        :: lambda1, lambda2
```

[1] This example can be found online at `https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`.

```
    integer                         :: v

    call lambda1%set( x, x+2 )
    call lambda2%set( x, x*2 )

    do v = 1,10
        write(*,*) v, lambda1%eval(v), lambda2%eval(v)
    enddo
end program
```

2 Laying the groundwork

Lambda expressions are possible in Fortran, but we need to provide some derived types and associated functions and subroutines to make it work. Here we will restrict ourselves to integer variables – extending this work to other data types is straightforward but perhaps somewhat tedious. Let us start by defining a derived type that can be used to store an expression tree, that is: we need to store the various operations and the operands instead of the resulting values:

```
    type lambda_integer
        integer                       :: operation
        integer                       :: value
        type(lambda_integer), pointer :: first  => null()
        type(lambda_integer), pointer :: second => null()
    end type lambda_integer
```

The type `lambda_integer` can contain an integer value (hence the component `value`) as well as a unary or binary operation, identified by the component `operation`[2] and the components `first` and `second`. Arithmetic operations on this derived type, like addition, are defined to store the expression:

```
function integer_add( x, y ) result(add)
    type(lambda_integer), intent(in), target  :: x
    type(lambda_integer), intent(in), target  :: y
    type(lambda_integer), pointer             :: add

    allocate( add )

    add%operation =  1
    add%first     => x
    add%second    => y
end function integer_add
```

[2] Using function pointers instead of integer values to select the operation would be more elegant.

(with similar versions for adding a `lambda_integer` variable to an ordinary integer). This way the compiler will unravel the expression and turn it into an expression tree.

We also need a routine to actually evaluate the expression as stored. For this we need to set the `lambda_integer` variables to some value, leading to code like:[3]

```
type(lambda_integer) :: x
type(lambda_integer) :: y
type(lambda_integer) :: expr

!
! Define the expression
!
call expr%set( x+2*y )

x = 1                   ! Set the values of the "free" variables that
y = 2                   ! occur in the expression

write(*,*) 'Sum of x and y = ', expr%eval()
```

where type-bound procedures have been used.[4].

The complication is that we do not really want to keep the original `x` and `y` variables around – they are important for the expression, but should be considered part of it. If we were to pass the expression contained in the variable `expr` to a subroutine, we would need to pass these variables too.

One solution is to wrap the pointer references via a second derived type:

```
type lambda_expression
    type(lambda_integer_pointer)      :: arg(4)     ! Arbitrary number
    type(lambda_integer)              :: operand(4)
    type(lambda_integer), pointer     :: expr
contains
    procedure :: set  => set_expression
    procedure :: eval => eval_expression
end type lambda_expression
```

and let the `set` method deal with these references:

```
!
! Implementation of the "set" method
!
subroutine set_expression( lambda, x, expr )
    class(lambda_expression)      :: lambda
    type(lambda_integer), target  :: x
```

[3]For conciseness, a lot of the details have been left out. See my book "Modern Fortran in practice".

[4]These are syntactic features not available in Fortran 90.

```
    type(lambda_integer), pointer :: expr

    type(lambda_integer_pointer), dimension(size(lambda%operand)) :: arg

    arg(1)%arg => x
    arg(2)%arg => null()
    arg(3)%arg => null()
    arg(4)%arg => null()

    !
    ! Correct the pointers to arguments
    !
    call correct_pointer( arg, lambda%operand, expr )

    allocate( lambda%expr, source=expr )
end subroutine set_expression
```

The dirty work is done by the `correct_pointer` routine:

- Scan the expression tree for references in the expression to the various arguments (in the above implementation, there is only one).
- Replace the references to the argument (`x`) by references *contained* in the elements of `arg`.

Finally, a copy of the expression that was passed is stored in the `lambda_expression` variable.[5]

3 The result

The actual implementation supports only a limited number of arithmetic operations for default integers and no logical operations at all. But the program as shown in the introduction produces the results you would expect:

```
           1           3           2
           2           4           4
           3           5           6
           4           6           8
           5           7          10
           6           8          12
           7           9          14
           8          10          16
           9          11          18
          10          12          20
```

[5] Note that this is a shallow copy – all pointers refer to items wholly contained in the lambda variable.

You could add a bit more "syntactic sugar" and write a routine `tabulate`, so that the program would look like:

```
program print_table
    use lambda_expressions
    type(lambda_integer) :: x

    call tabulate( 1, 10, x, [x+2, x*2] )
contains
subroutine tabulate( start, stop, x, expr_array )
    integer, intent(in)                                :: start, stop
    type(lambda_integer), intent(inout)                :: x
    type(lambda_integer), dimension(:), intent(inout) :: expr_array
    ...
end subroutine tabulate
end program
```

so as to hide almost all the details (identifying the "free" variable `x` cannot be hidden) and make the program accept any number of expressions for tabulation. That, however, is left as an exercise.

4 Leftovers

The code accompanying this note is far from complete – it is a mere demonstration that this is possible even without extending the syntax of the language.

Things that could be done to make it a bit more useful:

- Extend to more arithmetic operations and include logical operations
- Extend to include real data
- Consider derived types: can they be implemented in an elegant way or not?

The basic idea already appeared in my book "Modern Fortran in practice", Cambridge University Press, 2012.

The source code can be found at `https://sourceforge.net/p/flibs/svncode/HEAD/tree/trunk/experiments/lambda.f90`